PROGRAMOWANIE STRUKTUR CYFROWYCH

PicoBlaze
XILINX 8-bit Embedded Microcontroller
User Guide: UG129

© Dr inż. Ignacy Pardyka

UNIWERSYTET
JANA KOCHANOWSKIEGO
w Kielcach[1]

Rok akad. 2011/2012

[1] Instytut Fizyki, Zak-ad Informatyki, e-mail: ignacy.pardyka@ujk.edu.pl

1. Introduction
   - PicoBlaze
   - PicoBlaze Functional Blocks
2. PicoBlaze Interface Signals
3. Instruction Set
4. Interrupts
5. Scratchpad RAM
6. Input and Output Ports
   - Input Operations
   - Output Operations
7. Instruction Storage Configurations
8. PicoBlaze Development Tools
9. PicoBlaze Programming
10. PicoBlaze Instruction Codes
11. Related Materials and References

PicoBlaze

PicoBlaze User Guide

UG129 (v2.0) January 28, 2010

PicoBlaze Block Diagram

PicoBlaze Structure

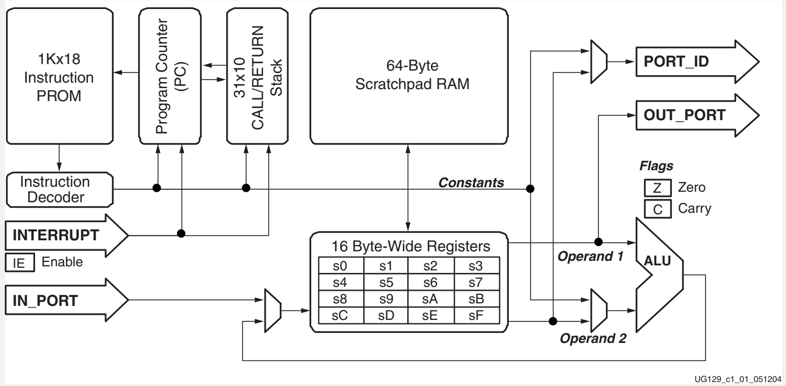


Figure 1-1: **PicoBlaze Embedded Microcontroller Block Diagram**

1 Introduction
- PicoBlaze
- PicoBlaze Functional Blocks

2 PicoBlaze Interface Signals

3 Instruction Set

4 Interrupts

5 Scratchpad RAM

6 Input and Output Ports
- Input Operations
- Output Operations

7 Instruction Storage Configurations

8 PicoBlaze Development Tools

9 PicoBlaze Programming

10 PicoBlaze Instruction Codes

11 Related Materials and References

PicoBlaze Interface

PicoBlaze Interface Connections

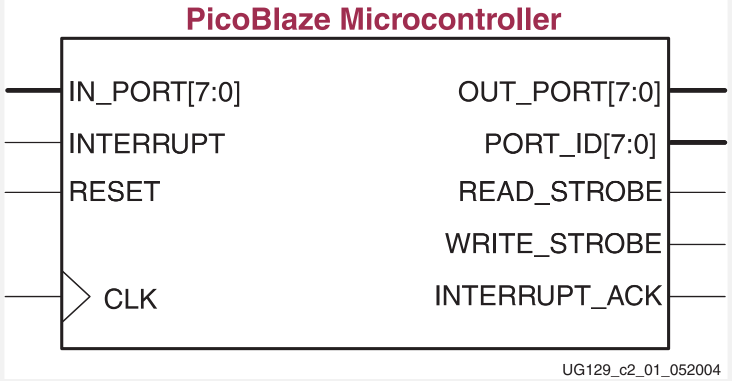
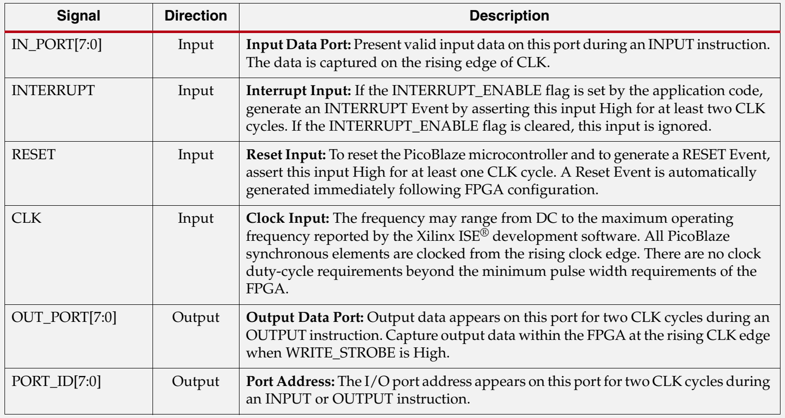


Figure 2-1: **PicoBlaze Interface Connections**

PicoBlaze Interface

PicoBlaze Interface Signals

Table 2-1: **PicoBlaze Interface Signal Descriptions**

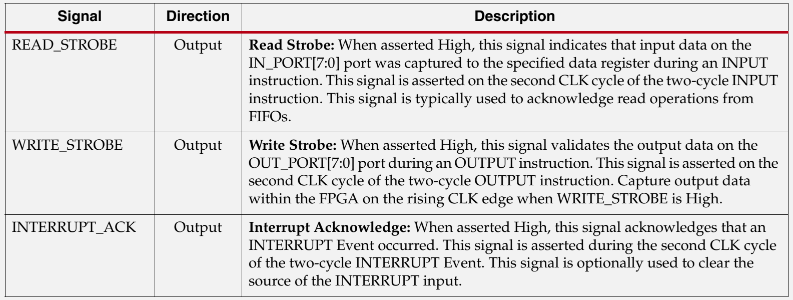


Signal	Direction	Description
IN_PORT[7:0]	Input	**Input Data Port:** Present valid input data on this port during an INPUT instruction. The data is captured on the rising edge of CLK.
INTERRUPT	Input	**Interrupt Input:** If the INTERRUPT_ENABLE flag is set by the application code, generate an INTERRUPT Event by asserting this input High for at least two CLK cycles. If the INTERRUPT_ENABLE flag is cleared, this input is ignored.
RESET	Input	**Reset Input:** To reset the PicoBlaze microcontroller and to generate a RESET Event, assert this input High for at least one CLK cycle. A Reset Event is automatically generated immediately following FPGA configuration.
CLK	Input	**Clock Input:** The frequency may range from DC to the maximum operating frequency reported by the Xilinx ISE® development software. All PicoBlaze synchronous elements are clocked from the rising clock edge. There are no clock duty-cycle requirements beyond the minimum pulse width requirements of the FPGA.
OUT_PORT[7:0]	Output	**Output Data Port:** Output data appears on this port for two CLK cycles during an OUTPUT instruction. Capture output data within the FPGA at the rising CLK edge when WRITE_STROBE is High.
PORT_ID[7:0]	Output	**Port Address:** The I/O port address appears on this port for two CLK cycles during an INPUT or OUTPUT instruction.

PicoBlaze Interface

PicoBlaze Interface Signals cont'd

Table 2-1: **PicoBlaze Interface Signal Descriptions *(Cont'd)***



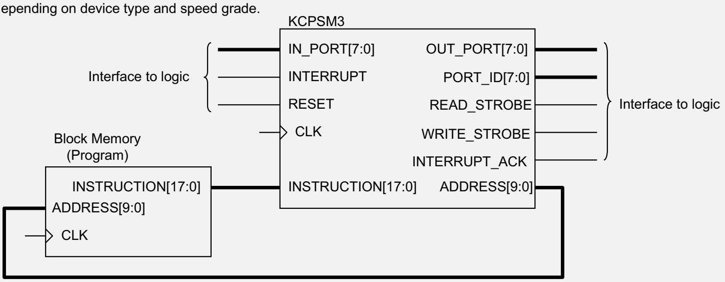
Signal	Direction	Description
READ_STROBE	Output	**Read Strobe:** When asserted High, this signal indicates that input data on the IN_PORT[7:0] port was captured to the specified data register during an INPUT instruction. This signal is asserted on the second CLK cycle of the two-cycle INPUT instruction. This signal is typically used to acknowledge read operations from FIFOs.
WRITE_STROBE	Output	**Write Strobe:** When asserted High, this signal validates the output data on the OUT_PORT[7:0] port during an OUTPUT instruction. This signal is asserted on the second CLK cycle of the two-cycle OUTPUT instruction. Capture output data within the FPGA on the rising CLK edge when WRITE_STROBE is High.
INTERRUPT_ACK	Output	**Interrupt Acknowledge:** When asserted High, this signal acknowledges that an INTERRUPT Event occurred. This signal is asserted during the second CLK cycle of the two-cycle INTERRUPT Event. This signal is optionally used to clear the source of the INTERRUPT input.

PicoBlaze

KCPSM3

KCPSM3 is a very simple 8-bit microcontroller primarily for the Spartan-3 devices but also suitable for use in Virtex-II and Virtex-IIPRO devices. Although it could be used for processing of data, it is most likely to be employed in applications requiring a complex, but non-time critical state machine. Hence it has the name of '(K)constant Coded Programmable State Machine'.

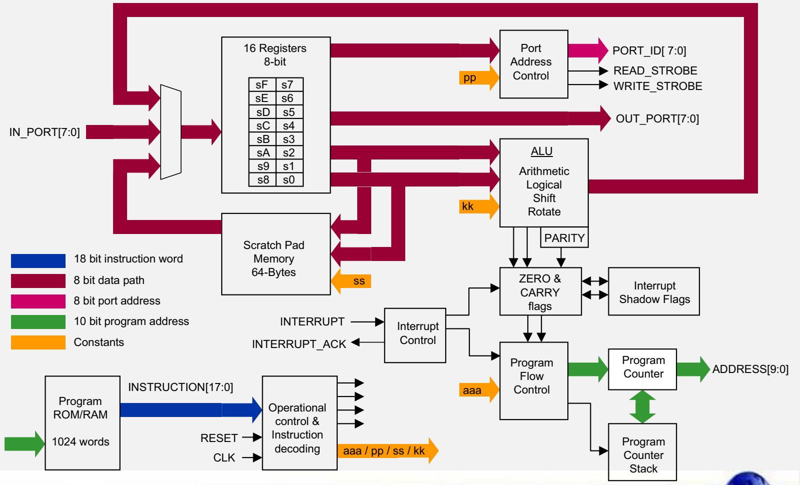
This revised version of popular KCPSM macro has still been developed with one dominant factor being held above all others - Size! The result is a microcontroller which occupies just **96 Spartan-3 Slices** which is just 5% of the XC3S200 device and less than 0.3% of the XC3S5000 device. Together with this small amount of logic, a single block RAM is used to form a ROM store for a program of up to 1024 instructions. Even with such size constraints, the performance is respectable at approximately **43 to 66 MIPS** depending on device type and speed grade.


KCPSM3
Interface to logic
IN_PORT[7:0]
INTERRUPT
RESET
CLK
OUT_PORT[7:0]
PORT_ID[7:0]
READ_STROBE
WRITE_STROBE
INTERRUPT_ACK
Interface to logic
Block Memory
(Program)
INSTRUCTION[17:0]
ADDRESS[9:0]
CLK
INSTRUCTION[17:0]
ADDRESS[9:0]

One of the most exciting features of the KCPSM3 is that it is totally embedded into the device and requires no external support. The very fact that ANY logic can be connected to the module inside the Spartan-3 or Virtex-II device means that any additional features can be added to provide ultimate flexibility. It is not so much what is inside the KCPSM3 module that makes it useful, but the environment in which it lives.

PicoBlaze

KCPSM3 Architecture


16 Registers
8-bit
sF s7
sE s6
sD s5
sC s4
sB s3
sA s2
s9 s1
s8 s0
Port
Address
Control
pp
PORT_ID[7:0]
READ_STROBE
WRITE_STROBE
OUT_PORT[7:0]
IN_PORT[7:0]
ALU
Arithmetic
Logical
Shift
Rotate
kk
PARITY
Scratch Pad
Memory
64-Bytes
ss
ZERO &
CARRY
flags
Interrupt
Shadow Flags
🟦 18 bit instruction word
🟥 8 bit data path
🟪 8 bit port address
🟩 10 bit program address
🟧 Constants
INTERRUPT
INTERRUPT_ACK
Interrupt
Control
Program
Flow
Control
aaa
Program
Counter
ADDRESS[9:0]
Program
Counter
Stack
Program
ROM/RAM
1024 words
INSTRUCTION[17:0]
Operational
control &
Instruction
decoding
RESET
CLK
aaa / pp / ss / kk

1. Introduction
   - PicoBlaze
   - PicoBlaze Functional Blocks
2. PicoBlaze Interface Signals
3. **Instruction Set**
4. Interrupts
5. Scratchpad RAM
6. Input and Output Ports
   - Input Operations
   - Output Operations
7. Instruction Storage Configurations
8. PicoBlaze Development Tools
9. PicoBlaze Programming
10. PicoBlaze Instruction Codes
11. Related Materials and References

PicoBlaze Instruction Set

Instructions

'X' and 'Y' refer to the definition of the storage registers 's' in the range 0 to F.
'kk' represents a constant value in the range 00 to FF.
'aaa' represents an address in the range 000 to 3FF.
'pp' represents a port address in the range 00 to FF.
'ss' represents an internal storage address in the range 00 to 3F.

Program Control Group

```
JUMP aaa
JUMP Z,aaa
JUMP NZ,aaa
JUMP C,aaa
JUMP NC,aaa

CALL aaa
CALL Z,aaa
CALL NZ,aaa
CALL C,aaa
CALL NC,aaa

RETURN
RETURN Z
RETURN NZ
RETURN C
RETURN NC
```

Note that call and return supports up to a stack depth of 31.

Arithmetic Group

```
ADD sX,kk
ADDCY sX,kk
SUB sX,kk
SUBCY sX,kk
COMPARE sX,kk

ADD sX,sY
ADDCY sX,sY
SUB sX,sY
SUBCY sX,sY
COMPARE sX,sY
```

Interrupt Group

```
RETURNI ENABLE
RETURNI DISABLE

ENABLE INTERRUPT
DISABLE INTERRUPT
```

Logical Group

```
LOAD sX,kk
AND sX,kk
OR sX,kk
XOR sX,kk
TEST sX,kk

LOAD sX,sY
AND sX,sY
OR sX,sY
XOR sX,sY
TEST sX,sY
```

Storage Group

```
STORE sX,ss
STORE sX,(sY)
FETCH sX,ss
FETCH sX,(sY)
```

Shift and Rotate Group

```
SR0 sX
SR1 sX
SRX sX
SRA sX
RR sX

SL0 sX
SL1 sX
SLX sX
SLA sX
RL sX
```

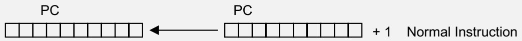
Input/Output Group

```
INPUT sX,pp
INPUT sX,(sY)
OUTPUT sX,pp
OUTPUT sX,(sY)
```

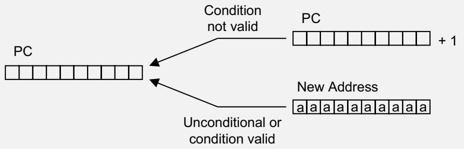
PicoBlaze Instruction Set

JUMP

Under normal conditions, the program counter (PC) increments to point to the next instruction. The address space is fixed to 1024 locations (000 to 3FF hex) and therefore the program counter is 10 bits wide. It is worth noting that the top of memory is 3FF hex and will increment to 000.



The JUMP instruction may be used to modify this sequence by specifying a new address. However, the JUMP instruction may be conditional. A conditional JUMP will only be performed if a test performed on either the ZERO flag or CARRY flag is valid. The JUMP instruction has no effect on the status of the flags.



```
JUMP aaa
JUMP Z,aaa
JUMP NZ,aaa
JUMP C,aaa
JUMP NC,aaa
```

Each JUMP instruction must specify the 10-bit address as a 3 digit hexadecimal value. The assembler supports labels to simplify this process.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0				a	a	a	a	a	a	a	a	a	a

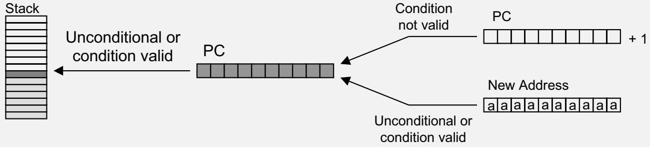
Bit 12 0 - UNCONDITIONAL
1 - CONDITIONAL

Bit 11	Bit 10	Condition
0	0	if Zero
0	1	if NOT Zero
1	0	if Carry
1	1	if NOT Carry

PicoBlaze Instruction Set

CALL

The CALL instruction is similar in operation to the JUMP instruction in that it will modify the normal program execution sequence by specifying a new address. The CALL instruction may also be conditional. In addition to supplying a new address, the CALL instruction also causes the current program counter (PC) value to be pushed onto the program counter stack. The CALL instruction has no effect on the status of the flags.



```
CALL aaa
CALL Z,aaa
CALL NZ,aaa
CALL C,aaa
CALL NC,aaa
```

The program counter stack supports a depth of 31 address values. This enables nested 'CALL' sequences to a depth of 31 levels to be performed. However, the stack will also be used during an interrupt operation and hence at least one of these levels should be reserved when interrupts are enabled. The stack is implemented as a separate cyclic buffer. When the stack becomes full, it simply overwrites the oldest value. Hence it is not necessary to reset the stack pointer when performing a software reset. This also explains why there are no instructions to control the stack and why no other memory needs to be reserved or provided for the stack.

Each CALL instruction must specify the 10-bit address as a 3 digit hexadecimal value. The assembler supports labels to simplify this process.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0				a	a	a	a	a	a	a	a	a	a

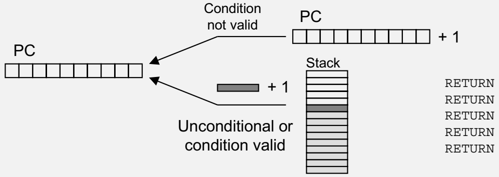
Bit 12 0 - UNCONDITIONAL
1 - CONDITIONAL

Bit 11	Bit 10	Condition
0	0	if Zero
0	1	if NOT Zero
1	0	if Carry
1	1	if NOT Carry

PicoBlaze Instruction Set

RETURN

The RETURN instruction is the complement to the CALL instruction. The RETURN instruction may also be conditional. In this case the new program counter (PC) value will be formed internally by incrementing the last value on the program address stack. This ensures that the program will execute the instruction following the CALL instruction which resulted in the subroutine. The RETURN instruction has no effect on the status of the flags.



```
RETURN
RETURN Z
RETURN NZ
RETURN C
RETURN NC
```

It is the responsibility of the programmer to ensure that a RETURN is only performed in response to a previous CALL instruction such that the program counter stack contains a valid address. The cyclic implementation of the stack will continue to provide values for RETURN instructions which can not be defined.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1				0	0	0	0	0	0	0	0	0	0

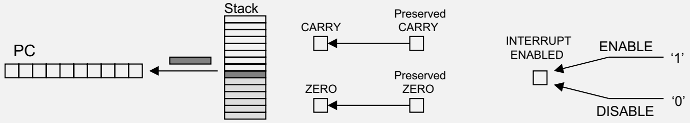
Bit 12 0 - UNCONDITIONAL
1 - CONDITIONAL

Bit 11	Bit 10	Condition
0	0	if Zero
0	1	if NOT Zero
1	0	if Carry
1	1	if NOT Carry

PicoBlaze Instruction Set

RETURNI

The RETURNI instruction is a special variation of the RETURN instruction which should be used to conclude an interrupt service routine. The RETURNI is unconditional and therefore will always load the program counter (PC) with the last address on the program counter stack (the address is not incremented in this case since the instruction at the address stored will need to be executed). The RETURNI instruction restores the flags to the condition they were in at the point of interrupt. The RETURNI also determines the future ability of interrupts using ENABLE and DISABLE as an operand.



It is the responsibility of the programmer to ensure that a RETURNI is only performed in response to an interrupt. Each RETURNI must specify if further interrupt is to be enabled or disabled.

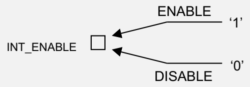
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RETURNI ENABLE	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RETURNI DISABLE	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

PicoBlaze Instruction Set

ENABLE/DISABLE INTERRUPT

These instructions are used to set and reset the INT_ENABLE flag. Before using ENABLE INTERRUPT a suitable interrupt routine must be associated with the interrupt address vector (located at address 3FF). Interrupts should never be enabled whilst performing an interrupt service routine.



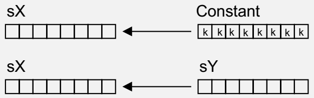
Interrupts are masked when the INT_ENABLE flag is low. This is the default state of the flag following device configuration or a KCPSM3 reset. The INT_ENABLE is also reset during an active interrupt.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ENABLE INTERRUPT	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
DISABLE INTERRUPT	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

PicoBlaze Instruction Set

LOAD

The LOAD instruction provides a method for specifying the contents of any register. The new value can be a constant, or the contents of any other register. The LOAD instruction has no effect on the status of the flags.



Since the LOAD instruction does not effect the flags it may be used to reorder and assign register contents at any stage of the program execution. The ability to assign a constant with no impact to the program size or performance means that the load instruction is the most obvious way to assign a value or clear a register.

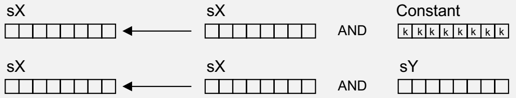
The first operand of a LOAD instruction must specify the register to be loaded as register 's' followed by a hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LOAD sX,kk	0	0	0	0	0	0	x	x	x	x	k	k	k	k	k	k	k	k
							sX				Constant							
LOAD sX,sY	0	0	0	0	0	1	x	x	x	x	y	y	y	y	0	0	0	0
							sX				sY							

PicoBlaze Instruction Set

AND

The AND instruction performs a bit-wise logical 'AND' operation between two operands. For example 00001111 AND 00110011 will produce the result 00000011. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The AND operation is useful for resetting bits of a register and performing tests on the contents (see also TEST instruction). The status of the ZERO flag will then control the flow of the program.



CARRY 0 ZERO ? Set if all bits of result are zero. Reset in all other cases.

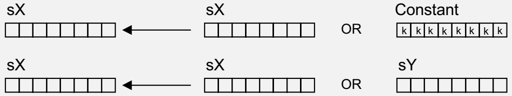
Each AND instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND sX,kk	0	0	1	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k
							sX				Constant							
AND sX,sY	0	0	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
							sX				sY							

PicoBlaze Instruction Set

OR

The OR instruction performs a bit-wise logical 'OR' operation between two operands. For example 00001111 OR 00110011 will produce the result 00111111. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. OR provides a way to force any bits of the specified register to be set which can be useful in forming control signals.



CARRY 0 ZERO ? Set if all bits of result are zero. Reset in all other cases.

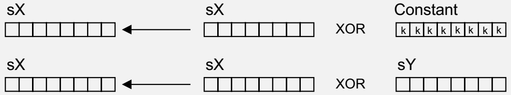
Each OR instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OR sX,kk	0	0	1	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k
							sX				Constant							
OR sX,sY	0	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
							sX				sY							

PicoBlaze Instruction Set

XOR

The XOR instruction performs a bit-wise logical 'XOR' operation between two operands. For example 00001111 XOR 00110011 will produce the result 00111100. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The XOR operation is useful for inverting bits contained in a register which is useful in forming control signals.



CARRY **0** ZERO **?** Set if all bits of result are zero. Reset in all other cases.

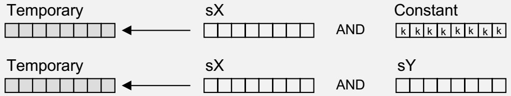
Each XOR instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XOR sX,kk	0	0	1	1	1	0	x	x	x	x	k	k	k	k	k	k	k	k
							sX				Constant							
XOR sX,sY	0	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
							sX				sY							

PicoBlaze Instruction Set

TEST

The TEST instruction performs a bit-wise logical 'AND' operation between two operands. Unlike the 'AND' instruction, the result of the operation is discarded and only the flags are affected. The ZERO flag is set if all bits of the temporary result are low. The CARRY flag is used to indicate the **ODD PARITY** of the temporary result. Parity checks typically involve a test of all bits, i.e. if the contents of 's5' = 3D (00111101), the execution of TEST s5,FF will set the CARRY flag indicating ODD parity. Bit testing is typically used to isolate a single bit. For example TEST s5,04 will test bit2 of the 's5' register which would set the CARRY flag if the bit is high (reset if the bit is low) and set the ZERO flag if the bit is low (reset if the bit is high).



CARRY **?** Set if there are an odd number of bits set to '1' in the temporary result. ZERO **?** Set if all bits of temporary result are zero. Reset in all other cases.

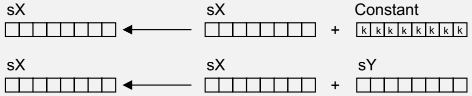
Each TEST instruction must specify the first operand register as 's' followed by a hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TEST sX,kk	0	1	0	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k
							sX				Constant							
TEST sX,sY	0	1	0	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
							sX				sY							

PicoBlaze Instruction Set

ADD

The ADD instruction performs an 8-bit addition of two operands. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. Note that this instruction does not use the CARRY as an input, and hence there is no need to condition the flags before use. The ability to specify any constant is useful in forming control sequences and counters.



CARRY **?** Set if result of addition exceeds FF. Reset in all other cases. ZERO **?** Set if all bits of result are zero. Reset in all other cases.

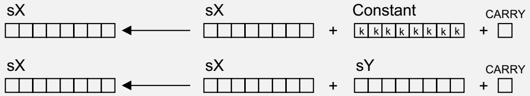
Each ADD instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD sX,kk	0	1	1	0	0	0	x	x	x	x	k	k	k	k	k	k	k	k
							sX				Constant							
ADD sX,sY	0	1	1	0	0	1	x	x	x	x	y	y	y	y	0	0	0	0
							sX				sY							

PicoBlaze Instruction Set

ADDCY

The ADDCY instruction performs an addition of two 8-bit operands together with the contents of the CARRY flag. The first operand is any register, and it is this register which will be assigned the result of the operation. A second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The ADDCY operation can be used in the formation of adder and counter processes exceeding 8 bits.



CARRY **?** Set if result of addition exceeds FF. Reset in all other cases. ZERO **?** Set if all bits of result are zero. Reset in all other cases.

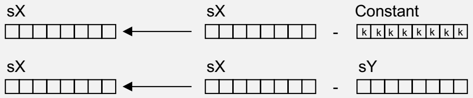
Each ADDCY instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDCY sX,kk	0	1	1	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k
							sX				Constant							
ADDCY sX,sY	0	1	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
							sX				sY							

PicoBlaze Instruction Set

SUB

The SUB instruction performs an 8-bit subtraction of two operands. The first operand is any register, and it is this register which will be assigned the result of the operation. The second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. Note that this instruction does not use the CARRY as an input, and hence there is no need to condition the flags before use. The CARRY flag indicates when an underflow has occurred. For example, if 's05' contains 27 hex and the instruction SUB s05,35 is performed, then the stored result will be F2 hex and the CARRY flag will be set.



CARRY ? Set if result is negative. Reset in all other cases.

ZERO ? Set if all bits of result are zero. Reset in all other cases.

Each SUB instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUB sX,kk	0	1	1	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k

sX Constant

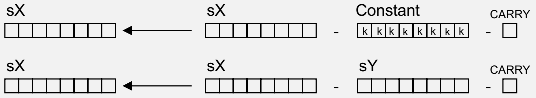
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUB sX,sY	0	1	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0

sX sY

PicoBlaze Instruction Set

SUBCY

The SUBCY instruction performs an 8-bit subtraction of two operands together with the contents of the CARRY flag. The first operand is any register, and it is this register which will be assigned the result of the operation. The second operand may also be any register or an 8-bit constant value. Flags will be effected by this operation. The SUBCY operation can be used in the formation of subtract and down counter processes exceeding 8 bits.



CARRY ? Set if result is negative. Reset in all other cases.

ZERO ? Set if all bits of result are zero. Reset in all other cases.

Each SUBCY instruction must specify the first operand register as 's' followed by a hexadecimal digit. This register will also form the destination for the result. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUBCY sX,kk	0	1	1	1	1	0	x	x	x	x	k	k	k	k	k	k	k	k

sX Constant

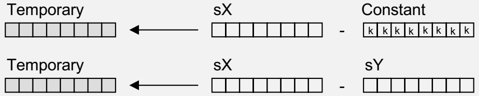
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUBCY sX,sY	0	1	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0

sX sY

PicoBlaze Instruction Set

COMPARE

The COMPARE instruction performs an 8-bit subtraction of two operands Unlike the 'SUB' instruction, the result of the operation is discarded and only the flags are affected. The ZERO flag is set when all the bits of the temporary result are low and indicates that both input operands were identical. The CARRY flag indicates when an underflow has occurred and indicates that the second operand was larger than the first. For example, if 's05' contains 27 hex and the instruction COMPARE s05,35 is performed, then the CARRY flag will be set (35>27) and the ZERO flag will be reset (35≠27).



CARRY ? Set if 'sY' or 'kk' is **greater than** 'sX'. Reset in all other cases.

ZERO ? Set if operands are **equal**. Reset in all other cases.

Each COMPARE instruction must specify the first operand register as 's' followed by a hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPARE sX,kk	0	1	0	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k

sX Constant

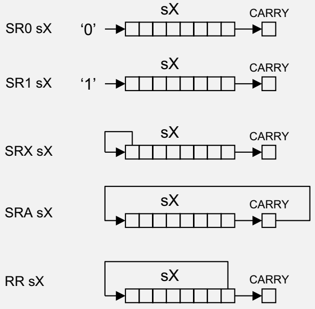
	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMPARE sX,sY	0	1	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0

sX sY

PicoBlaze Instruction Set

SR0, SR1, SRX, SRA, RR

The shift and rotate right group all modify the contents of a single register. All instructions in the group have an effect on the flags.



SR0 sX: ZERO ? Set if all bits of result are zero. Reset in all other cases.

SR1 sX: ZERO 0

SRX sX: ZERO ? Set if all bits of result are zero. Reset in all other cases.

SRA sX: ZERO ? Set if all bits of result are zero. Reset in all other cases.

RR sX: ZERO ? Set if all bits of result are zero. Reset in all other cases.

Each instruction must specify the register as 's' followed by a hexadecimal digit. The assembler supports register naming to simplify the process.

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	x	x	x	x	0	0	0	0	1			

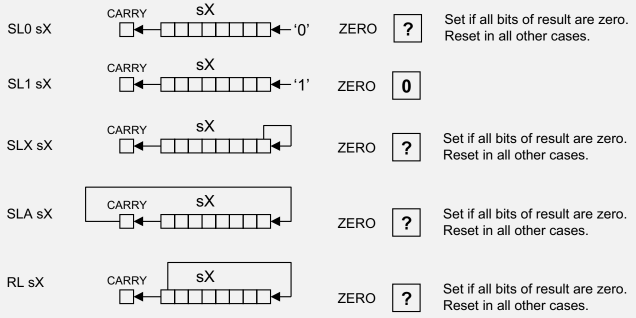
sX

Bit 2	Bit 1	Bit 0	Instruction
1	1	0	SR0 sX
1	1	1	SR1 sX
0	1	0	SRX sX
0	0	0	SRA sX
1	0	0	RR sX

PicoBlaze Instruction Set

SL0, SL1, SLX, SLA, RL

The shift and rotate left group all modify the contents of a single register. All instructions in the group have an effect on the flags.



SL0 sX — CARRY — sX — '0' — ZERO ?
Set if all bits of result are zero.
Reset in all other cases.

SL1 sX — CARRY — sX — '1' — ZERO 0

SLX sX — CARRY — sX — ZERO ?
Set if all bits of result are zero.
Reset in all other cases.

SLA sX — CARRY — sX — ZERO ?
Set if all bits of result are zero.
Reset in all other cases.

RL sX — CARRY — sX — ZERO ?
Set if all bits of result are zero.
Reset in all other cases.

Each instruction must specify the register as 's' followed by a hexadecimal digit. The assembler supports register naming to simplify the process.

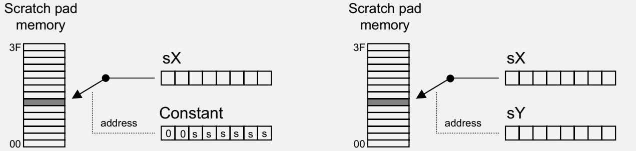
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	x	x	x	x	0	0	0	0	0			
						sX											

Bit 2	Bit 1	Bit 0	Instruction
1	1	0	SL0 sX
1	1	1	SL1 sX
1	0	0	SLX sX
0	0	0	SLA sX
0	1	0	RL sX

PicoBlaze Instruction Set

STORE

The STORE instruction enables the contents of any register to be transferred to the 64-byte internal scratch pad memory. The storage address (in the range 00 to 3F) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.



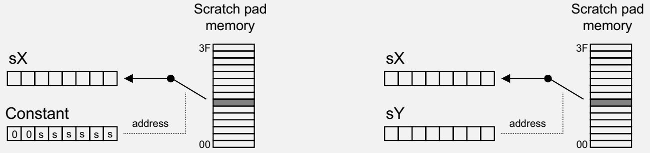
Each STORE instruction must specify the source register as 's' followed by a hexadecimal digit. It must then specify the storage address using a register value in a similar way or specify a 6-bit constant storage address using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process. Although the assembler will reject constants greater than 3F, it is the responsibility of the programmer to ensure that the value of 'sY' is within the address range.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STORE sX,PP	1	0	1	1	1	0	x	x	x	x	0	0	s	s	s	s	s	s
							sX						Constant address					
STORE sX,(sY)	1	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
							sX				sY							

PicoBlaze Instruction Set

FETCH

The FETCH instruction enables data held in the 64-byte internal scratch pad memory to be transferred any of the internal registers. The storage address (in the range 00 to 3F) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.



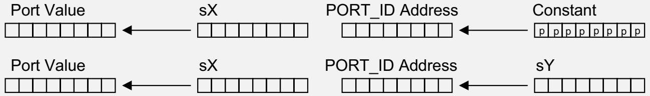
Each FETCH instruction must specify the destination register as 's' followed by a hexadecimal digit. It must then specify the storage address using a register value in a similar way or specify a 6-bit constant storage address using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process. Although the assembler will reject constants greater than 3F, it is the responsibility of the programmer to ensure that the value of 'sY' is within the address range.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FETCH sX,PP	0	0	0	1	1	0	x	x	x	x	0	0	s	s	s	s	s	s
							sX						Constant address					
FETCH sX,(sY)	0	0	0	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
							sX				sY							

PicoBlaze Instruction Set

OUTPUT

The OUTPUT instruction enables the contents of any register to be transferred to logic external to KCPSM3. The port address (in the range 00 to FF) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.



The user interface logic is required to decode the PORT_ID port address value and capture the data provided on the OUT_PORT. The WRITE_STROBE is set during an output operation (see 'READ and WRITE STROBES'), and should be used to clock enable the capture register or write enable a RAM (see 'Design of Output Ports').

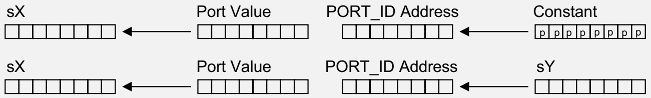
Each OUTPUT instruction must specify the source register as 's' followed by a hexadecimal digit. It must then specify the output port address using a register value in a similar way or specify an 8-bit constant port identifier using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUTPUT sX,PP	1	0	1	1	0	0	x	x	x	x	p	p	p	p	p	p	p	p
							sX				Constant PORT_ID							
OUTPUT sX,(sY)	1	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
							sX				sY							

PicoBlaze Instruction Set

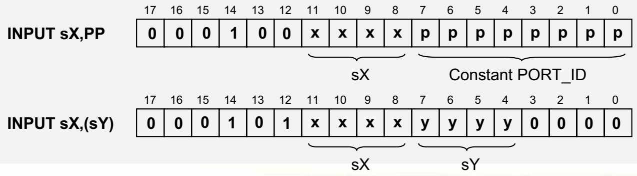
INPUT

The INPUT instruction enables data values external to KCPSM3 to be transferred into any one of the internal registers. The port address (in the range 00 to FF) can be defined by a constant value or indirectly as the contents of any other register. The Flags are not affected by this operation.



The user interface logic is required to decode the PORT_ID port address value and supply the correct data to the IN_PORT. The READ_STROBE is set during an input operation (see ‘READ and WRITE STROBES’), but it is not always necessary for the interface logic to decode this strobe. However, it can be useful for determining when data has been read, such as when reading a FIFO buffer (see ‘Design of Input Ports’).

Each INPUT instruction must specify the destination register as ‘s’ followed by a hexadecimal digit. It must then specify the input port address using a register value in a similar way or specify an 8-bit constant port identifier using 2 hexadecimal digits. The assembler supports register naming and constant labels to simplify the process.

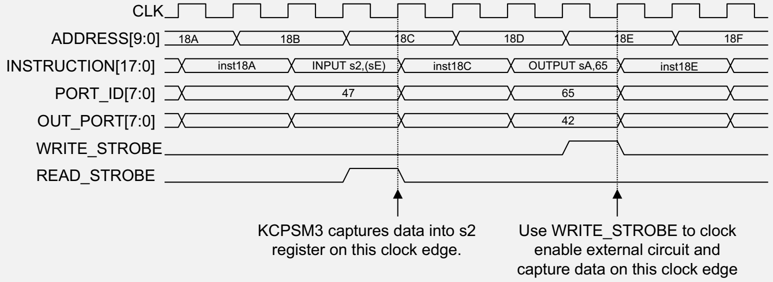


	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INPUT sX,PP	0	0	0	1	0	0	x	x	x	x	p	p	p	p	p	p	p	p
							sX				Constant PORT_ID							
INPUT sX,(sY)	0	0	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
							sX				sY							

PicoBlaze Instruction Set

READ and WRITE STROBES

These pulses are used by external circuits to confirm input and output operations. In the waveforms below, it is assumed that the content of register sE is 47, and the content of register sA is 42.



KCPSM3 captures data into s2 register on this clock edge.

Use WRITE_STROBE to clock enable external circuit and capture data on this clock edge

PORT_ID[7:0] is valid for 2 clock cycles providing additional time for external decoding logic and enabling the connection of synchronous RAM. The WRITE_STROBE is provided on the second clock cycle to confirm an active write by KCPSM3. In most cases, the READ_STROBE will not be utilised by the external decoding logic, but again occurs in the second cycle and indicates the actual clock edge on which data is read into the specified register.

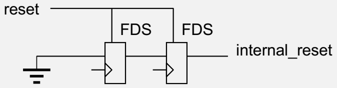
Note for timing critical designs, your timing specifications can allow 2 clock cycles for PORT_ID and data paths, and only the strobes need to be constrained to a single clock cycle. Ideally, a pipeline register can be inserted where possible (see ‘Design of Input Ports’, ‘Design of Output Ports’ and ‘Connecting Memory’).

PicoBlaze

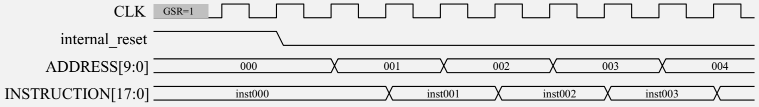
KCPSM3 RESET

KCPSM3 contains an internal reset control circuit to ensure the correct start up of KCPSM3 following device configuration or global reset (GSR). This reset can also be activated within your design.

The KCPSM3 reset is sampled synchronous to the clock and used to form a controlled internal reset signal which is distributed locally as required. A small ‘filter’ circuit (see right) ensures that the release of the internal reset is clean and controlled.

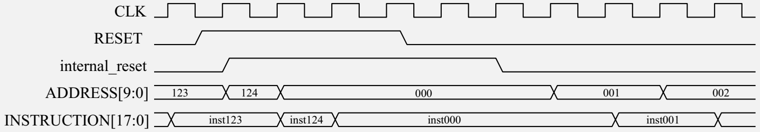


Release of Reset after configuration.



Application of user reset input

The reset input can be tied to logic ‘0’ if not required and the ‘filter’ will still be used to ensure correct power-up sequence.



Moving Data

Moving Data

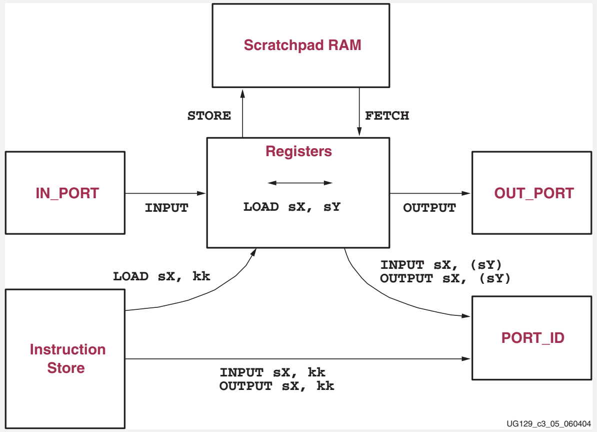
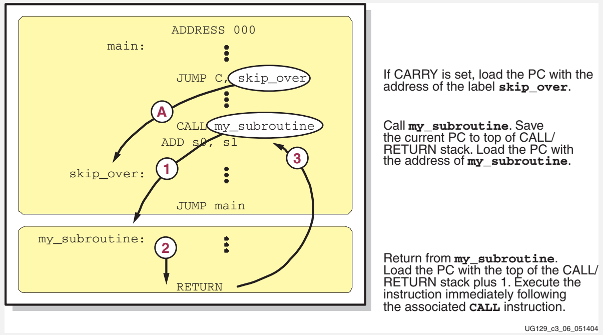


Figure 3-26: **Data Movement Instructions**

JUMP

JUMP and CALL/RETURN



```
                ADDRESS 000
main:
        JUMP C, skip_over
        CALL my_subroutine
        ADD s0, s1
skip_over:
        JUMP main

my_subroutine:
        RETURN
```

If CARRY is set, load the PC with the address of the label **skip_over**.

Call **my_subroutine**. Save the current PC to top of CALL/ RETURN stack. Load the PC with the address of **my_subroutine**.

Return from **my_subroutine**. Load the PC with the top of the CALL/ RETURN stack plus 1. Execute the instruction immediately following the associated **CALL** instruction.

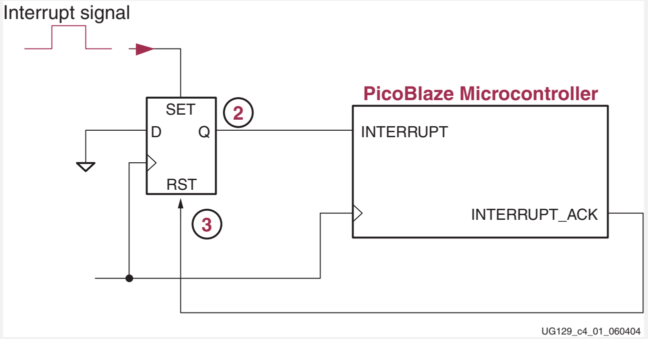
UG129_c3_06_051404

Figure 3-27: **Example JUMP and CALL/RETURN Procedures**

1. Introduction
   - PicoBlaze
   - PicoBlaze Functional Blocks
2. PicoBlaze Interface Signals
3. Instruction Set
4. Interrupts
5. Scratchpad RAM
6. Input and Output Ports
   - Input Operations
   - Output Operations
7. Instruction Storage Configurations
8. PicoBlaze Development Tools
9. PicoBlaze Programming
10. PicoBlaze Instruction Codes
11. Related Materials and References

Interrupts

Interrupt Logic

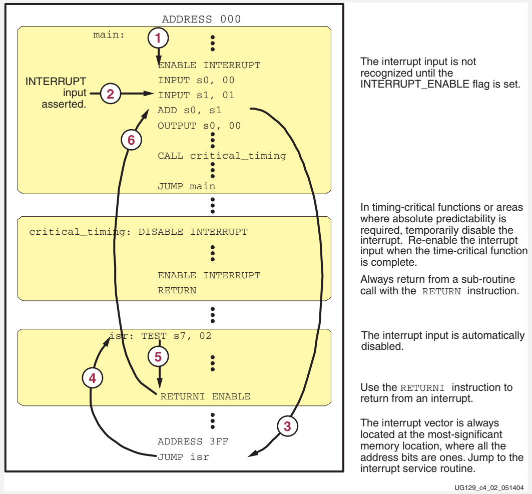


UG129_c4_01_060404

Figure 4-1: **Simple Interrupt Logic**

Interrupts

Interrupt Flow



```
                ADDRESS 000
main:
                ENABLE INTERRUPT
                INPUT s0, 00
                INPUT s1, 01
                ADD s0, s1
                OUTPUT s0, 00
                CALL critical_timing
                JUMP main

critical_timing: DISABLE INTERRUPT
                ENABLE INTERRUPT
                RETURN

        isr: TEST s7, 02
                RETURNI ENABLE

                ADDRESS 3FF
                JUMP isr
```

The interrupt input is not recognized until the INTERRUPT_ENABLE flag is set.

In timing-critical functions or areas where absolute predictability is required, temporarily disable the interrupt. Re-enable the interrupt input when the time-critical function is complete.

Always return from a sub-routine call with the RETURN instruction.

The interrupt input is automatically disabled.

Use the RETURNI instruction to return from an interrupt.

The interrupt vector is always located at the most-significant memory location, where all the address bits are ones. Jump to the interrupt service routine.

UG129_c4_02_051404

Figure 4-2: **Example Interrupt Flow**

Interrupts

Interrupt Timing Diagram

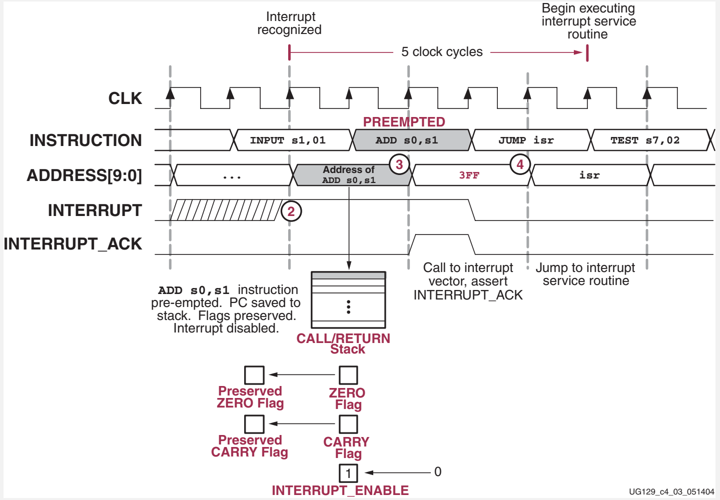
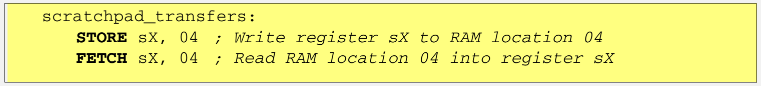


Figure 4-3: **Interrupt Timing Diagram**

1. Introduction
   - PicoBlaze
   - PicoBlaze Functional Blocks
2. PicoBlaze Interface Signals
3. Instruction Set
4. Interrupts
5. Scratchpad RAM
6. Input and Output Ports
   - Input Operations
   - Output Operations
7. Instruction Storage Configurations
8. PicoBlaze Development Tools
9. PicoBlaze Programming
10. PicoBlaze Instruction Codes
11. Related Materials and References

Address Modes

Direct Addressing

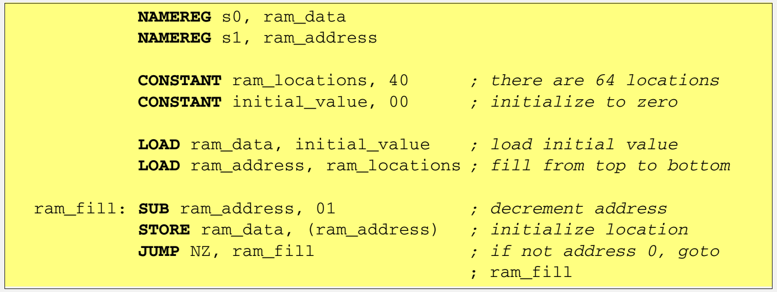


```
scratchpad_transfers:
    STORE sX, 04  ; Write register sX to RAM location 04
    FETCH sX, 04  ; Read RAM location 04 into register sX
```

Figure 5-1: **Directly Addressing Scratchpad RAM Locations**

Address Modes

Indirect Addressing



```
           NAMEREG s0, ram_data
           NAMEREG s1, ram_address

           CONSTANT ram_locations, 40      ; there are 64 locations
           CONSTANT initial_value, 00      ; initialize to zero

           LOAD ram_data, initial_value    ; load initial value
           LOAD ram_address, ram_locations ; fill from top to bottom

ram_fill:  SUB ram_address, 01             ; decrement address
           STORE ram_data, (ram_address)   ; initialize location
           JUMP NZ, ram_fill               ; if not address 0, goto
                                           ; ram_fill
```

Figure 5-2: **Indirect Addressing Initializes All of RAM with a Simple Subroutine**

Scratchpad RAM

FETCH Operation

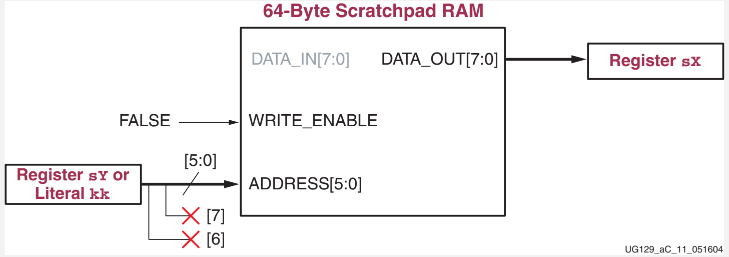


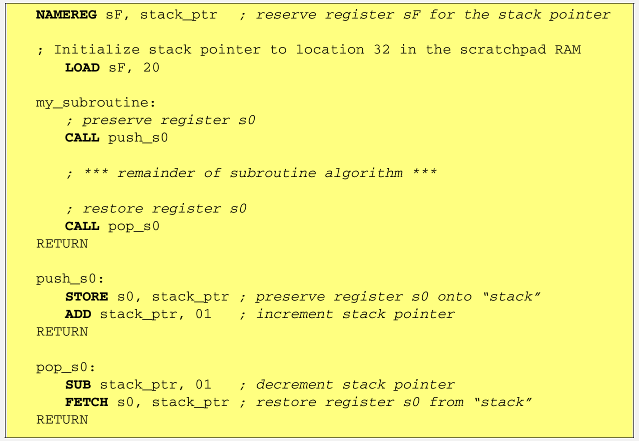
Figure C-5: **FETCH Operation**

Examples

```
FETCH sX, (sY) ; Read scratchpad RAM location specified by the
               ; contents of register sY into register sX

FETCH sX, kk   ; Read scratchpad RAM location specified by the
               ; immediate constant kk into register sX
```

Stack Operations in Program

Stack in RAM



```
NAMEREG sF, stack_ptr   ; reserve register sF for the stack pointer

; Initialize stack pointer to location 32 in the scratchpad RAM
   LOAD sF, 20

my_subroutine:
   ; preserve register s0
   CALL push_s0

   ; *** remainder of subroutine algorithm ***

   ; restore register s0
   CALL pop_s0
RETURN

push_s0:
   STORE s0, stack_ptr ; preserve register s0 onto "stack"
   ADD stack_ptr, 01   ; increment stack pointer
RETURN

pop_s0:
   SUB stack_ptr, 01   ; decrement stack pointer
   FETCH s0, stack_ptr ; restore register s0 from "stack"
RETURN
```

Figure 5-4: **Use Scratchpad RAM to Emulate PUSH and POP Stack Operations**

1 Introduction
- PicoBlaze
- PicoBlaze Functional Blocks

2 PicoBlaze Interface Signals

3 Instruction Set

4 Interrupts

5 Scratchpad RAM

6 Input and Output Ports
- Input Operations
- Output Operations

7 Instruction Storage Configurations

8 PicoBlaze Development Tools

9 PicoBlaze Programming

10 PicoBlaze Instruction Codes

11 Related Materials and References

Input Operation

Input Operation

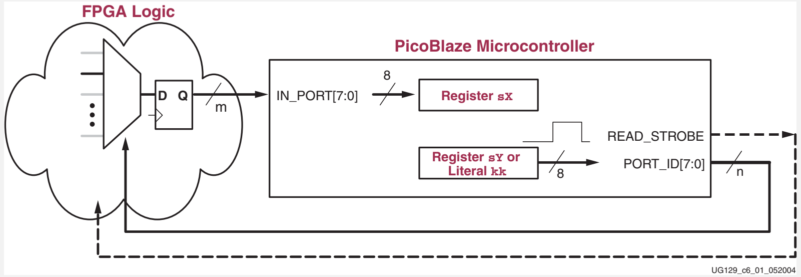
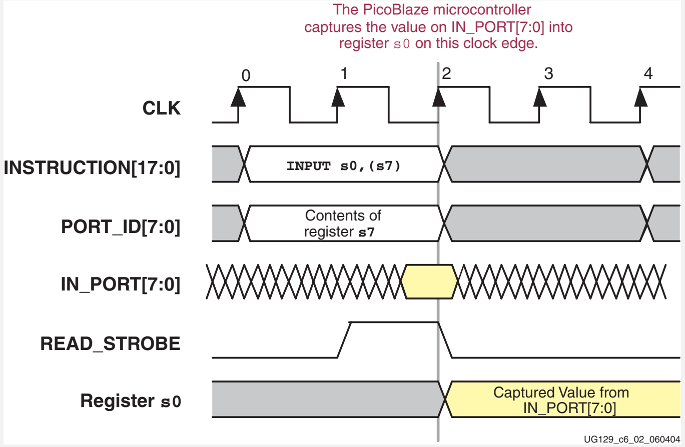


Figure 6-1: **INPUT Operation and FPGA Interface Logic**

Input Operation

Port Timing

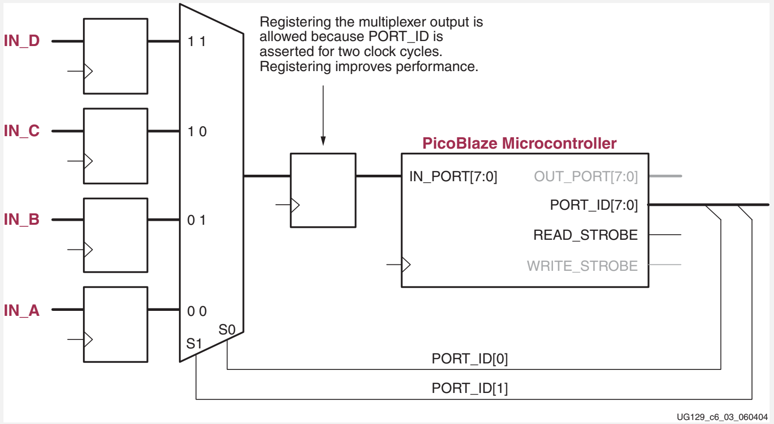


The PicoBlaze microcontroller captures the value on IN_PORT[7:0] into register `s0` on this clock edge.

Figure 6-2: **Port Timing for INPUT Instruction**

Multiple Input Sources

Multiplex to Form a Single IN_PORT Port

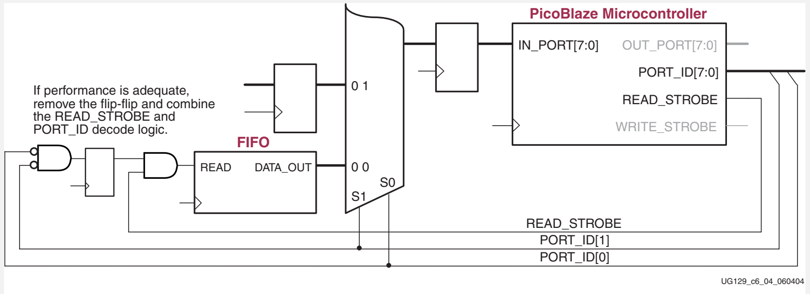


Registering the multiplexer output is allowed because PORT_ID is asserted for two clock cycles. Registering improves performance.

Figure 6-3: **Multiplex Multiple Input Sources to Form a Single IN_PORT Port**

Input from FIFO

Using READ_STROBE



If performance is adequate, remove the flip-flip and combine the READ_STROBE and PORT_ID decode logic.

Figure 6-4: **READ_STROBE Indicates a Successful INPUT Operation**

Output Operation

FPGA Interface

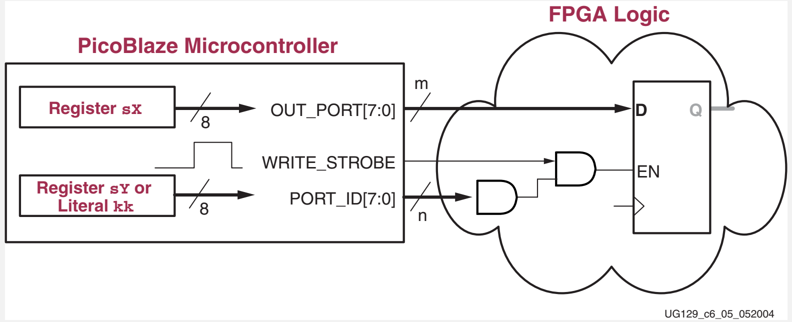


Figure 6-5: **OUTPUT Operation and FPGA Interface**

Output Operation

Port Timing

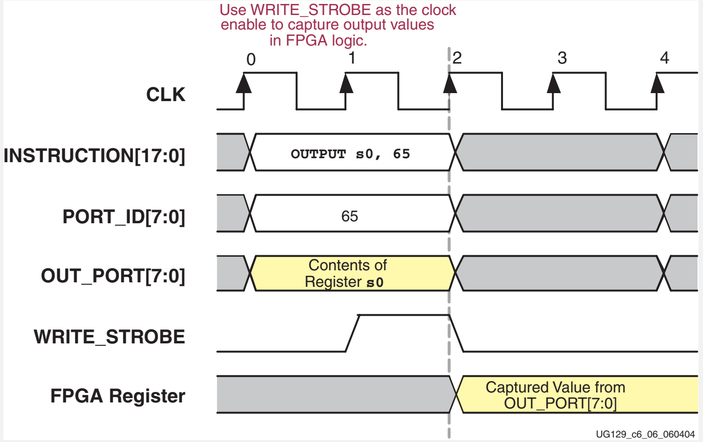


Figure 6-6: **Port Timing for OUTPUT Instruction**

Output Destinations

Address Decoding

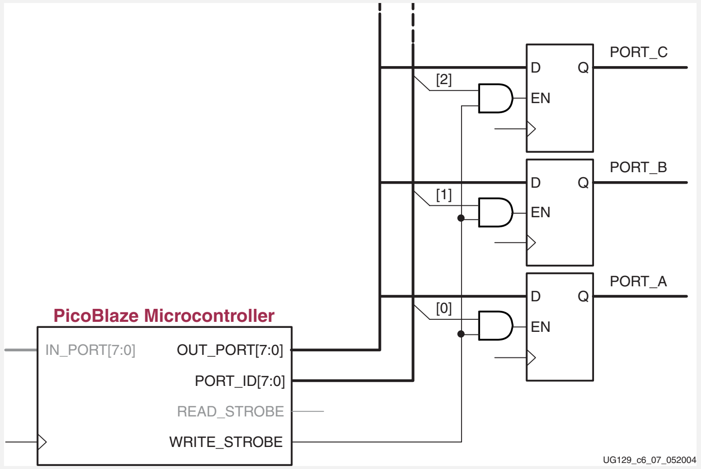
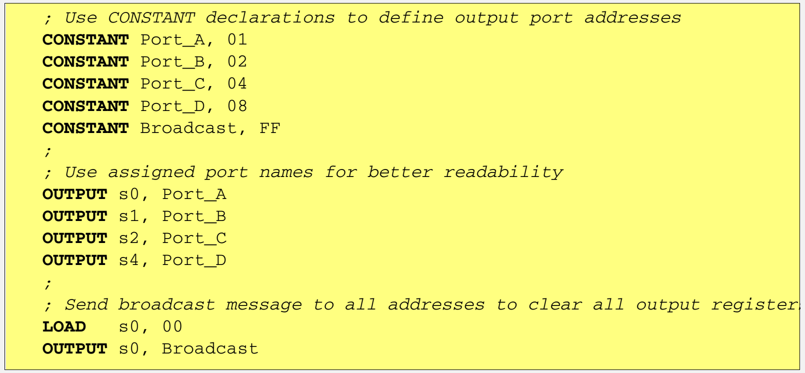


Figure 6-7: **Simple Address Decoding for Designs with Few Output Destinations**

Output Destinations

Output Ports Declaration



```
; Use CONSTANT declarations to define output port addresses
CONSTANT Port_A, 01
CONSTANT Port_B, 02
CONSTANT Port_C, 04
CONSTANT Port_D, 08
CONSTANT Broadcast, FF
;
; Use assigned port names for better readability
OUTPUT s0, Port_A
OUTPUT s1, Port_B
OUTPUT s2, Port_C
OUTPUT s4, Port_D
;
; Send broadcast message to all addresses to clear all output register
LOAD   s0, 00
OUTPUT s0, Broadcast
```

Figure 6-8: **Use CONSTANT Directives to Declare Output Port Addresses**

1 Introduction
- PicoBlaze
- PicoBlaze Functional Blocks

2 PicoBlaze Interface Signals

3 Instruction Set

4 Interrupts

5 Scratchpad RAM

6 Input and Output Ports
- Input Operations
- Output Operations

7 **Instruction Storage Configurations**

8 PicoBlaze Development Tools

9 PicoBlaze Programming

10 PicoBlaze Instruction Codes

11 Related Materials and References

Instruction Store

Standard Configuration

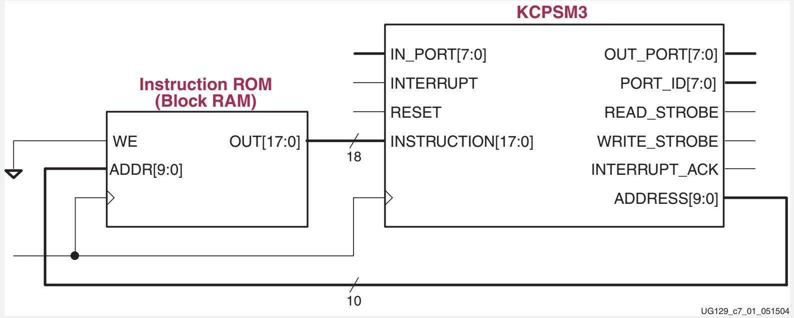


Figure 7-1: **Standard Implementation using a Single 1Kx18 Block RAM as the Instruction Store**

Instruction Store

Loading the Program

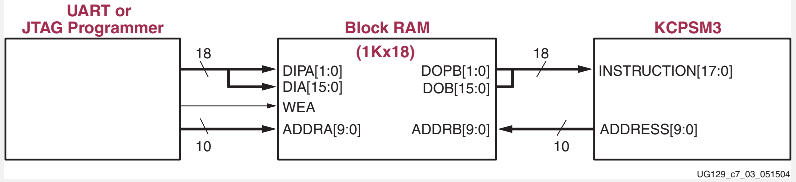


Figure 7-3: **Standard Configuration with UART or JTAG Program Loader**

Two Microcontrollers

Sharing a Common Program Memory

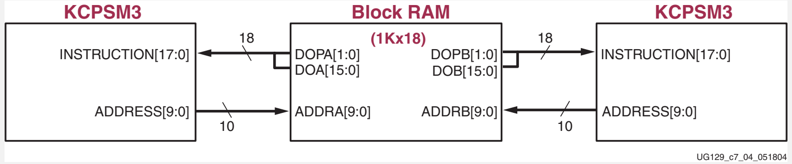
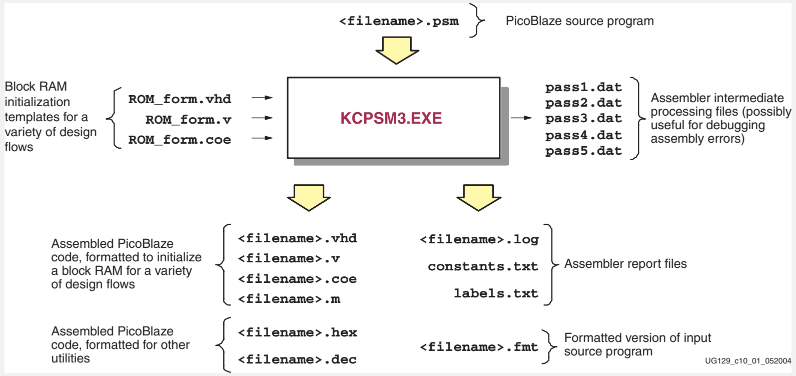


Figure 7-4: **Two PicoBlaze Microcontrollers Sharing a Common Code Image**

1. Introduction
   - PicoBlaze
   - PicoBlaze Functional Blocks
2. PicoBlaze Interface Signals
3. Instruction Set
4. Interrupts
5. Scratchpad RAM
6. Input and Output Ports
   - Input Operations
   - Output Operations
7. Instruction Storage Configurations
8. **PicoBlaze Development Tools**
9. PicoBlaze Programming
10. PicoBlaze Instruction Codes
11. Related Materials and References

KCPSM3

Assembler Files

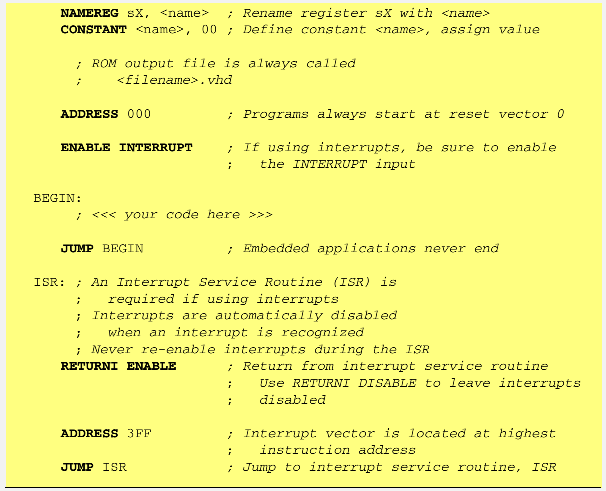


<filename>.psm
PicoBlaze source program
Block RAM initialization templates for a variety of design flows
ROM_form.vhd
ROM_form.v
ROM_form.coe
KCPSM3.EXE
pass1.dat
pass2.dat
pass3.dat
pass4.dat
pass5.dat
Assembler intermediate processing files (possibly useful for debugging assembly errors)
Assembled PicoBlaze code, formatted to initialize a block RAM for a variety of design flows
<filename>.vhd
<filename>.v
<filename>.coe
<filename>.m
<filename>.log
constants.txt
labels.txt
Assembler report files
Assembled PicoBlaze code, formatted for other utilities
<filename>.hex
<filename>.dec
<filename>.fmt
Formatted version of input source program
UG129_c10_01_052004

Figure 10-1: **KCPSM3 Assembler Files**

KCPSM3

Application Program Template

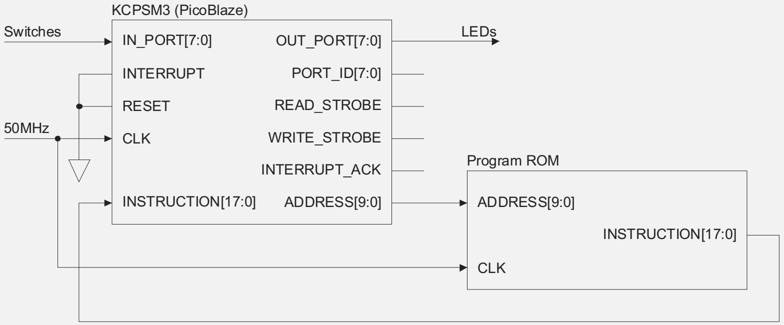


```
    NAMEREG sX, <name>   ; Rename register sX with <name>
    CONSTANT <name>, 00 ; Define constant <name>, assign value

      ; ROM output file is always called
      ;    <filename>.vhd

    ADDRESS 000          ; Programs always start at reset vector 0

    ENABLE INTERRUPT     ; If using interrupts, be sure to enable
                         ;   the INTERRUPT input

BEGIN:
      ; <<< your code here >>>

    JUMP BEGIN           ; Embedded applications never end

ISR: ; An Interrupt Service Routine (ISR) is
     ;   required if using interrupts
     ; Interrupts are automatically disabled
     ;   when an interrupt is recognized
     ; Never re-enable interrupts during the ISR
    RETURNI ENABLE       ; Return from interrupt service routine
                         ;   Use RETURNI DISABLE to leave interrupts
                         ;   disabled

    ADDRESS 3FF          ; Interrupt vector is located at highest
                         ;   instruction address
    JUMP ISR             ; Jump to interrupt service routine, ISR
```

Figure B-1: **PicoBlaze Application Program Template for KCPSM3 Assembler**

KCPSM3

Simple Base System



Using PicoBlaze Microcontroller Module

Sample Toplevel Verilog Module

```
module toplevel(
    input clk,
    output [7:0] leds,
    input [7:0] sliders,
    input [3:0] buttons,
    ...
     );

     testprg progrom(.address(address), .instruction(instruction), .clk(clk));


kcpsm3 pblaze
     (
  .address(address),
     .instruction(instruction),
     .port_id(port_id),
     .write_strobe(write_strobe),
     .out_port(out_port),
     .read_strobe(read_strobe),
     .in_port(in_port_reg),
     .interrupt(1'b0),
     .interrupt_ack(1'bZ),
     .reset(reset),
     .clk(clk)
  );

  ....
```

Instruction Memory Module

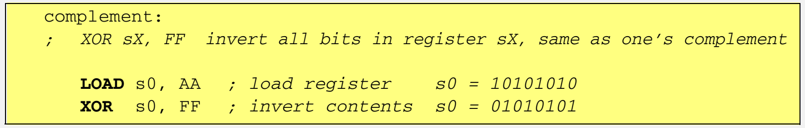
Generated by KCPSM3 Assembler

```
module testprg (address, instruction, clk);
input [9:0] address;
input clk;
output [17:0] instruction;
RAMB16_S18 ram_1024_x_18(
 .DI  (16'h0000),
 .DIP  (2'b00),
 .EN (1'b1),
 .WE (1'b0),
 .SSR (1'b0),
 .CLK (clk),
 .ADDR (address),
 .DO (instruction[15:0]),
 .DOP (instruction[17:16]))
/*synthesis
init_00 = "410740074004C6064601C5014502C405C304A40083010013C4050400C3040311"
init_01 = "00000000000000000000000000000000A00054148201000E0205A000500F5010"
init_02 = "0000000000000000000000000000000000000000000000000000000000000000"

...
endmodule
```

1. Introduction
   - PicoBlaze
   - PicoBlaze Functional Blocks
2. PicoBlaze Interface Signals
3. Instruction Set
4. Interrupts
5. Scratchpad RAM
6. Input and Output Ports
   - Input Operations
   - Output Operations
7. Instruction Storage Configurations
8. PicoBlaze Development Tools
9. PicoBlaze Programming
10. PicoBlaze Instruction Codes
11. Related Materials and References

Using PicoBlaze Instructions

Complementing a Register Value

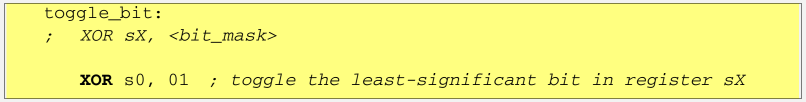


```
complement:
;   XOR sX, FF  invert all bits in register sX, same as one's complement

    LOAD s0, AA   ; load register    s0 = 10101010
    XOR  s0, FF   ; invert contents  s0 = 01010101
```

Figure 3-2: **Complementing a Register Value**

Using PicoBlaze Instructions

Inverting an Individual Bit Location



```
toggle_bit:
;   XOR sX, <bit_mask>

    XOR s0, 01  ; toggle the least-significant bit in register sX
```

Figure 3-3: **Inverting an Individual Bit Location**

Using PicoBlaze Instructions

Clearing a Register and Setting the ZERO Flag

```
XOR sX, sX  ; clear register sX, set ZERO flag
```

Figure 3-4: **Clearing a Register and Setting the ZERO Flag**

Using PicoBlaze Instructions

Clearing a Register without Modifying the ZERO Flag

```
LOAD sX,00  ; clear register sX, ZERO flag unaffected
```

Figure 3-5: **Clearing a Register without Modifying the ZERO Flag**

Using PicoBlaze Instructions

Set Bit

```
set_bit:
;   OR sX, <bit_mask>

    OR s0, 01 ; set bit 0 of register s0
```

Figure 3-6: **16-Setting a Bit Location**

Using PicoBlaze Instructions

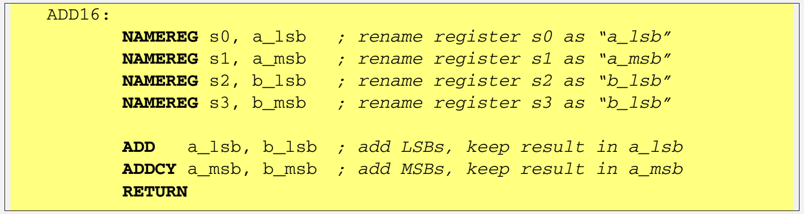
Clear Bit

```
clear_bit:
;   AND sX, <bit_mask>

    AND s0, FE  ; clear bit 0 of register s0
```

Figure 3-7: **Clearing a Bit Location**

Using PicoBlaze Instructions

16-Bit Addition

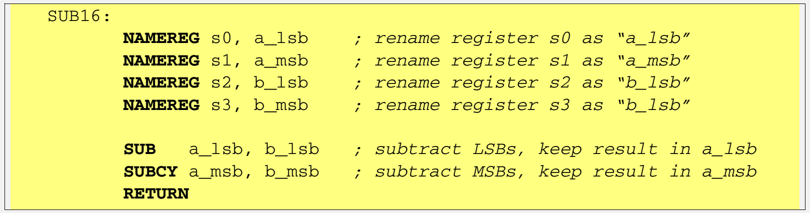


```
ADD16:
        NAMEREG s0, a_lsb   ; rename register s0 as “a_lsb”
        NAMEREG s1, a_msb   ; rename register s1 as “a_msb”
        NAMEREG s2, b_lsb   ; rename register s2 as “b_lsb”
        NAMEREG s3, b_msb   ; rename register s3 as “b_lsb”

        ADD    a_lsb, b_lsb ; add LSBs, keep result in a_lsb
        ADDCY a_msb, b_msb  ; add MSBs, keep result in a_msb
        RETURN
```

Figure 3-8: **16-Bit Addition Using ADD and ADDCY Instructions**

Using PicoBlaze Instructions

16-Bit Subtraction

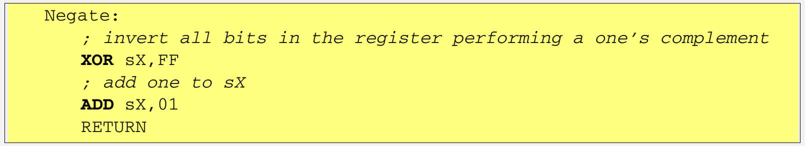


```
SUB16:
        NAMEREG s0, a_lsb     ; rename register s0 as “a_lsb”
        NAMEREG s1, a_msb     ; rename register s1 as “a_msb”
        NAMEREG s2, b_lsb     ; rename register s2 as “b_lsb”
        NAMEREG s3, b_msb     ; rename register s3 as “b_lsb”

        SUB    a_lsb, b_lsb   ; subtract LSBs, keep result in a_lsb
        SUBCY a_msb, b_msb    ; subtract MSBs, keep result in a_msb
        RETURN
```

Figure 3-9: **16-Bit Subtraction Using SUB and SUBCY Instructions**

Using PicoBlaze Instructions

Destructive Negate

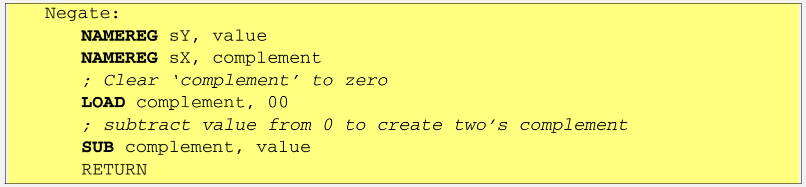


```
Negate:
    ; invert all bits in the register performing a one’s complement
    XOR sX,FF
    ; add one to sX
    ADD sX,01
    RETURN
```

Figure 3-12: **Destructive Negate (2’s Complement) Function Overwrites Original Value**

Using PicoBlaze Instructions

Non-destructive Negate



```
Negate:
    NAMEREG sY, value
    NAMEREG sX, complement
    ; Clear ‘complement’ to zero
    LOAD complement, 00
    ; subtract value from 0 to create two’s complement
    SUB complement, value
    RETURN
```

Figure 3-13: **Non-destructive Negate Function Preserves Original Value**

Using PicoBlaze Instructions

Hardware Multiplier

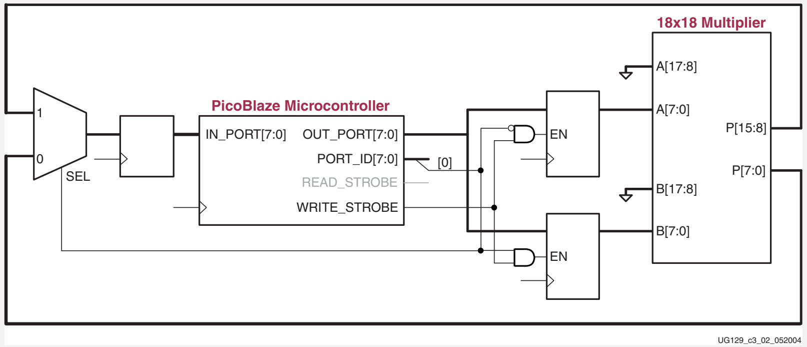
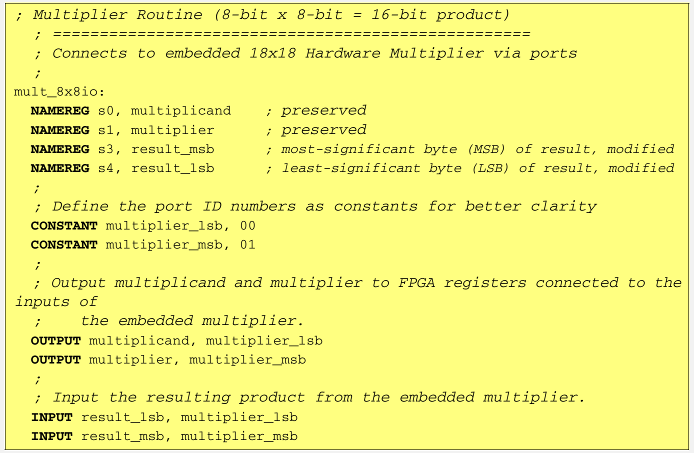


Figure 3-15: **8-bit by 8-bit Hardware Multiplier Using the FPGA's 18x18 Multipliers**

Using PicoBlaze Instructions

Assembler Program for Hardware Multiplier



```
; Multiplier Routine (8-bit x 8-bit = 16-bit product)
  ; ==================================================
  ; Connects to embedded 18x18 Hardware Multiplier via ports
  ;
mult_8x8io:
  NAMEREG s0, multiplicand    ; preserved
  NAMEREG s1, multiplier      ; preserved
  NAMEREG s3, result_msb      ; most-significant byte (MSB) of result, modified
  NAMEREG s4, result_lsb      ; least-significant byte (LSB) of result, modified
  ;
  ; Define the port ID numbers as constants for better clarity
  CONSTANT multiplier_lsb, 00
  CONSTANT multiplier_msb, 01
  ;
  ; Output multiplicand and multiplier to FPGA registers connected to the
inputs of
  ;    the embedded multiplier.
  OUTPUT multiplicand, multiplier_lsb
  OUTPUT multiplier, multiplier_msb
  ;
  ; Input the resulting product from the embedded multiplier.
  INPUT result_lsb, multiplier_lsb
  INPUT result_msb, multiplier_msb
```

Figure 3-16: **8-bit by 8-bit Multiply Routine Using Hardware Multiplier**

1 Introduction
- PicoBlaze
- PicoBlaze Functional Blocks

2 PicoBlaze Interface Signals

3 Instruction Set

4 Interrupts

5 Scratchpad RAM

6 Input and Output Ports
- Input Operations
- Output Operations

7 Instruction Storage Configurations

8 PicoBlaze Development Tools

9 PicoBlaze Programming

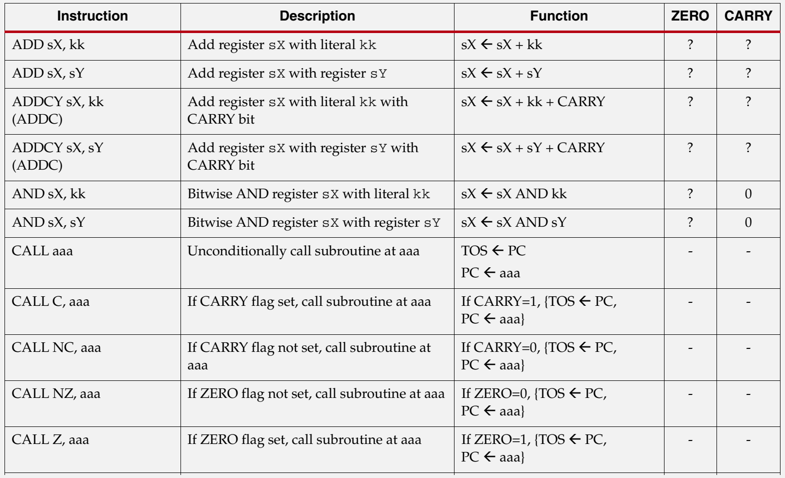
10 PicoBlaze Instruction Codes

11 Related Materials and References

PicoBlaze Instruction Set

Instructions

Table 3-1: **PicoBlaze Instruction Set (alphabetical listing)**



Instruction	Description	Function	ZERO	CARRY
ADD sX, kk	Add register sX with literal kk	sX ← sX + kk	?	?
ADD sX, sY	Add register sX with register sY	sX ← sX + sY	?	?
ADDCY sX, kk (ADDC)	Add register sX with literal kk with CARRY bit	sX ← sX + kk + CARRY	?	?
ADDCY sX, sY (ADDC)	Add register sX with register sY with CARRY bit	sX ← sX + sY + CARRY	?	?
AND sX, kk	Bitwise AND register sX with literal kk	sX ← sX AND kk	?	0
AND sX, sY	Bitwise AND register sX with register sY	sX ← sX AND sY	?	0

CALL aaa	Unconditionally call subroutine at aaa	TOS ← PC
		PC ← aaa
			-	-

CALL C, aaa	If CARRY flag set, call subroutine at aaa	If CARRY=1, {TOS ← PC, PC ← aaa}	-	-
CALL NC, aaa	If CARRY flag not set, call subroutine at aaa	If CARRY=0, {TOS ← PC, PC ← aaa}	-	-
CALL NZ, aaa	If ZERO flag not set, call subroutine at aaa	If ZERO=0, {TOS ← PC, PC ← aaa}	-	-
CALL Z, aaa	If ZERO flag set, call subroutine at aaa	If ZERO=1, {TOS ← PC, PC ← aaa}	-	-

PicoBlaze Instruction Set

Instructions

COMPARE sX, kk (COMP)	Compare register sX with literal kk. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If sX=kk, ZERO ← 1 If sX<kk, CARRY ← 1	?	?
COMPARE sX, sY (COMP)	Compare register sX with register sY. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If sX=sY, ZERO ← 1 If sX<sY, CARRY ← 1	?	?
DISABLE INTERRUPT (DINT)	Disable interrupt input	INTERRUPT_ENABLE ← 0	-	-

PicoBlaze Instruction Set

Instructions

Instruction	Description	Function	ZERO	CARRY
ENABLE INTERRUPT (EINT)	Enable interrupt input	INTERRUPT_ENABLE ← 1	-	-

Interrupt Event	Asynchronous interrupt input. Preserve flags and PC. Clear INTERRUPT_ENABLE flag. Jump to interrupt vector at address 3FF.	Preserved ZERO ← ZERO
		Preserved CARRY ← CARRY
		INTERRUPT_ENABLE ← 0
		TOS ← PC
		PC ← 3FF
			-	-

FETCH sX, (sY) (FETCH sX, sY)	Read scratchpad RAM location pointed to by register sY into register sX	sX ← RAM[(sY)]	-	-
FETCH sX, ss	Read scratchpad RAM location ss into register sX	sX ← RAM[ss]	-	-

INPUT sX, (sY)
(IN sX, sY)
	Read value on input port location pointed to by register sY into register sX	PORT_ID ← sY
		sX ← IN_PORT
			-	-

INPUT sX, pp
(IN)
	Read value on input port location pp into register sX	PORT_ID ← pp
		sX ← IN_PORT
			-	-

PicoBlaze Instruction Set

Instructions

JUMP aaa	Unconditionally jump to aaa	PC ← aaa	-	-
JUMP C, aaa	If CARRY flag set, jump to aaa	If CARRY=1, PC ← aaa	-	-
JUMP NC, aaa	If CARRY flag not set, jump to aaa	If CARRY=0, PC ← aaa	-	-
JUMP NZ, aaa	If ZERO flag not set, jump to aaa	If ZERO=0, PC ← aaa	-	-
JUMP Z, aaa	If ZERO flag set, jump to aaa	If ZERO=1, PC ← aaa	-	-
LOAD sX, kk	Load register sX with literal kk	sX ← kk	-	-
LOAD sX, sY	Load register sX with register sY	sX ← sY	-	-
OR sX, kk	Bitwise OR register sX with literal kk	sX ← sX OR kk	?	0
OR sX, sY	Bitwise OR register sX with register sY	sX ← sX OR sY	?	0

OUTPUT sX, (sY)
(OUT sX, sY)
	Write register sX to output port location pointed to by register sY	PORT_ID ← sY
		OUT_PORT ← sX
			-	-

OUTPUT sX, pp
(OUT sX, pp)
	Write register sX to output port location pp	PORT_ID ← pp
		OUT_PORT ← sX
			-	-

PicoBlaze Instruction Set

Instructions

RETURN (RET)	Unconditionally return from subroutine	PC ← TOS+1	-	-
RETURN C (RET C)	If CARRY flag set, return from subroutine	If CARRY=1, PC ← TOS+1	-	-
RETURN NC (RET NC)	If CARRY flag not set, return from subroutine	If CARRY=0, PC ← TOS+1	-	-
RETURN NZ (RET NZ)	If ZERO flag not set, return from subroutine	If ZERO=0, PC ← TOS+1	-	-
RETURN Z (RET Z)	If ZERO flag set, return from subroutine	If ZERO=1, PC ← TOS+1	-	-

PicoBlaze Instruction Set

Instructions

Instruction	Description	Function	ZERO	CARRY

RETURNI DISABLE
(RETI DISABLE)
	Return from interrupt service routine. Interrupt remains disabled.	PC ← TOS
		ZERO ← Preserved ZERO
		CARRY ← Preserved CARRY
		INTERRUPT_ENABLE ← 0
			?	?

RETURNI ENABLE
(RETI ENABLE)
	Return from interrupt service routine. Re-enable interrupt.	PC ← TOS
		ZERO ← Preserved ZERO
		CARRY ← Preserved CARRY
		INTERRUPT_ENABLE ← 1
			?	?

RL sX	Rotate register sX left	sX ← {sX[6:0],sX[7]}
		CARRY ← sX[7]
			?	?

RR sX	Rotate register sX right	sX ← {sX[0],sX[7:1]}
		CARRY ← sX[0]
			?	?

PicoBlaze Instruction Set

Instructions

SL0 sX	Shift register sX left, zero fill	sX ← {sX[6:0],0}
		CARRY ← sX[7]
			?	?

SL1 sX	Shift register sX left, one fill	sX ← {sX[6:0],1}
		CARRY ← sX[7]
			0	?

SLA sX	Shift register sX left through all bits, including CARRY	sX ← {sX[6:0],CARRY}
		CARRY ← sX[7]
			?	?

SLX sX	Shift register sX left. Bit sX[0] is unaffected.	sX ← {sX[6:0],sX[0]}
		CARRY ← sX[7]
			?	?

SR0 sX	Shift register sX right, zero fill	sX ← {0,sX[7:1]}
		CARRY ← sX[0]
			?	?

SR1 sX	Shift register sX right, one fill	sX ← {1,sX[7:1]}
		CARRY ← sX[0]
			0	?

SRA sX	Shift register sX right through all bits, including CARRY	sX ← {CARRY,sX[7:1]}
		CARRY ← sX[0]
			?	?

SRX sX	Arithmetic shift register sX right. Sign extend sX. Bit sX[7] Is unaffected.	sX ← {sX[7],sX[7:1]}
		CARRY ← sX[0]
			?	?

PicoBlaze Instruction Set

Instructions

STORE sX, (sY) (STORE sX, sY)	Write register sX to scratchpad RAM location pointed to by register sY	RAM[(sY)] ← sX	-	-
STORE sX, ss	Write register sX to scratchpad RAM location ss	RAM[ss] ← sX	-	-
SUB sX, kk	Subtract literal kk from register sX	sX ← sX – kk	?	?
SUB sX, sY	Subtract register sY from register sX	sX ← sX – sY	?	?
SUBCY sX, kk (SUBC)	Subtract literal kk from register sX with CARRY (borrow)	sX ← sX – kk - CARRY	?	?
SUBCY sX, sY (SUBC)	Subtract register sY from register sX with CARRY (borrow)	sX ← sX – sY - CARRY	?	?

PicoBlaze Instruction Set

Instructions

Instruction	Description	Function	ZERO	CARRY
TEST sX, kk	Test bits in register sX against literal kk. Update CARRY and ZERO flags. Registers are unaffected.	If (sX AND kk) = 0, ZERO ← 1 CARRY ← odd parity of (sX AND kk)	?	?
TEST sX, sY	Test bits in register sX against register sX. Update CARRY and ZERO flags. Registers are unaffected.	If (sX AND sY) = 0, ZERO ← 1 CARRY ← odd parity of (sX AND kk)	?	?
XOR sX, kk	Bitwise XOR register sX with literal kk	sX ← sX XOR kk	?	0
XOR sX, sY	Bitwise XOR register sX with register sY	sX ← sX XOR sY	?	0

PicoBlaze Instruction Set

Instructions

sX = One of 16 possible register locations ranging from s0 through sF or specified as a literal

sY = One of 16 possible register locations ranging from s0 through sF or specified as a literal

aaa = 10-bit address, specified either as a literal or a three-digit hexadecimal value ranging from 000 to 3FF or a labeled location

kk = 8-bit immediate constant, specified either as a literal or a two-digit hexadecimal value ranging from 00 to FF or specified as a literal

pp = 8-bit port address, specified either as a literal or a two-digit hexadecimal value ranging from 00 to FF or specified as a literal

ss = 6-bit scratchpad RAM address, specified either as a literal or a two-digit hexadecimal value ranging from 00 to 3F or specified as a literal

RAM[n] = Contents of scratchpad RAM at location n

TOS = Value stored at Top Of Stack

PicoBlaze Instructions

Instruction Codes

Table D-1: **PicoBlaze Instruction Codes**

Instruction	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD sX,kk	0	1	1	0	0	0	x	x	x	x	k	k	k	k	k	k	k	k
ADD sX,sY	0	1	1	0	0	1	x	x	x	x	y	y	y	y	0	0	0	0
ADDCY sX,kk	0	1	1	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k
ADDCY sX,sY	0	1	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
AND sX,kk	0	0	1	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k
AND sX,sY	0	0	1	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
CALL	1	1	0	0	0	0	0	0	a	a	a	a	a	a	a	a	a	a
CALL C	1	1	0	0	0	1	1	0	a	a	a	a	a	a	a	a	a	a
CALL NC	1	1	0	0	0	1	1	1	a	a	a	a	a	a	a	a	a	a
CALL NZ	1	1	0	0	0	1	0	1	a	a	a	a	a	a	a	a	a	a
CALL Z	1	1	0	0	0	1	0	0	a	a	a	a	a	a	a	a	a	a

PicoBlaze Instructions

Instruction Codes

COMPARE sX,kk	0	1	0	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k
COMPARE sX,sY	0	1	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
DISABLE INTERRUPT	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ENABLE INTERRUPT	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
FETCH sX, ss	0	0	0	1	1	0	x	x	x	x	0	0	s	s	s	s	s	s
FETCH sX,(sY)	0	0	0	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
INPUT sX,(sY)	0	0	0	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
INPUT sX,pp	0	0	0	1	0	0	x	x	x	x	p	p	p	p	p	p	p	p
JUMP	1	1	0	1	0	0	0	0	a	a	a	a	a	a	a	a	a	a
JUMP C	1	1	0	1	0	1	1	0	a	a	a	a	a	a	a	a	a	a
JUMP NC	1	1	0	1	0	1	1	1	a	a	a	a	a	a	a	a	a	a
JUMP NZ	1	1	0	1	0	1	0	1	a	a	a	a	a	a	a	a	a	a
JUMP Z	1	1	0	1	0	1	0	0	a	a	a	a	a	a	a	a	a	a

PicoBlaze Instructions

Instruction Codes

Table D-1: **PicoBlaze Instruction Codes *(Cont'd)***

Instruction	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LOAD sX,kk	0	0	0	0	0	0	x	x	x	x	k	k	k	k	k	k	k	k
LOAD sX,sY	0	0	0	0	0	1	x	x	x	x	y	y	y	y	0	0	0	0
OR sX,kk	0	0	1	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k
OR sX,sY	0	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
OUTPUT sX,(sY)	1	0	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
OUTPUT sX,pp	1	0	1	1	0	0	x	x	x	x	p	p	p	p	p	p	p	p
RETURN	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
RETURN C	1	0	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
RETURN NC	1	0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0
RETURN NZ	1	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0
RETURN Z	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
RETURNI DISABLE	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RETURNI ENABLE	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

PicoBlaze Instructions

Instruction Codes

RL sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	0	1	0
RR sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	1	0	0
SL0 sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	1	0
SL1 sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	1	1
SLA sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	0	0	0
SLX sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	0	1	0	0
SR0 sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	1	1	0
SR1 sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	1	1	1
SRA sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	0	0	0
SRX sX	1	0	0	0	0	0	x	x	x	x	0	0	0	0	1	0	1	0
STORE sX, ss	1	0	1	1	1	0	x	x	x	x	0	0	s	s	s	s	s	s
STORE sX,(sY)	1	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0

PicoBlaze Instructions

Instruction Codes

SUB sX,kk	0	1	1	1	0	0	x	x	x	x	k	k	k	k	k	k	k	k
SUB sX,sY	0	1	1	1	0	1	x	x	x	x	y	y	y	y	0	0	0	0
SUBCY sX,kk	0	1	1	1	1	0	x	x	x	x	k	k	k	k	k	k	k	k
SUBCY sX,sY	0	1	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0
TEST sX,kk	0	1	0	0	1	0	x	x	x	x	k	k	k	k	k	k	k	k
TEST sX,sY	0	1	0	0	1	1	x	x	x	x	y	y	y	y	0	0	0	0
XOR sX,kk	0	0	1	1	1	0	x	x	x	x	k	k	k	k	k	k	k	k
XOR sX,sY	0	0	1	1	1	1	x	x	x	x	y	y	y	y	0	0	0	0

1 Introduction
- PicoBlaze
- PicoBlaze Functional Blocks

2 PicoBlaze Interface Signals

3 Instruction Set

4 Interrupts

5 Scratchpad RAM

6 Input and Output Ports
- Input Operations
- Output Operations

7 Instruction Storage Configurations

8 PicoBlaze Development Tools

9 PicoBlaze Programming

10 PicoBlaze Instruction Codes

11 Related Materials and References

Related Materials and References

Bibliography

1. **PicoBlaze 8-bit Embedded Microcontroller**
   Download PicoBlaze reference designs and additional files.
   http://www.xilinx.com/ipcenter/processor_central/picoblaze
2. Mediatronix pBlazIDE Integrated Development Environment for PicoBlaze
   http://www.mediatronix.com/pBlazeIDE.htm
3. *Xilinx System Generator User Guide*: "Designing PicoBlaze Microcontroller Applications"
   http://www.xilinx.com/support/sw_manuals/sysgen_ug.pdf
4. MicroBlaze 32-bit Soft Processor Core
   http://www.xilinx.com/microblaze
5. UG331: *Spartan-3 Generation FPGA User Guide*: Chapter 8, "Using Dedicated Multiplexers"
   http://www.xilinx.com/support/documentation/user_guides/ug331.pdf
6. *XST User Guide*: Chapter 9, "Mixed Language Support"
   http://toolbox.xilinx.com/docsan/xilinx10/books/docs/xst/xst.pdf